Introduction to PANDAS

CS 3753 Data Science

Prof. Weining Zhang

Comments on Homework Assignments

1. Name your program notebook file lastname-firstname-hwkX
2. Include all files needed for running
3. Separate the problems in different code cells

Topics

- Series and DataFrame
  - Structure creation and alteration
  - indexing, index object,
- Support for data processing
  - Reindexing
  - Data selection
  - Arithmetic operations and function mapping
  - Sorting, ranking
  - Descriptive statistics
  - Hierarchical indexing

Overview of pandas Package

- The Pandas package (http://pandas.pydata.org/pandas-docs/stable/index.html) provids fast, flexible, and expressive data structures: Series and DataFrame, representing one- and two-dimensional tables

```
import pandas as pd
```

- Many functions are provided to work with DataFrame
  - Change structures
  - Update values
  - Statistics
  - Plot

pandas DataFrame vs numpy ndarray

- DataFrame differs from NumPy array in that
  - Elements can have arbitrary types
  - Both rows and columns have explicit index, by number as well as by name
  - More flexible and complex ways to slice, index, change the table structure
  - Database type of table operations, group-by, aggaregate, join,

In [ ]:
```
%matplotlib inline
from __future__ import division
import os
import matplotlib.pyplot as plt
plt.rc('figure', figsize=(10, 6))
from numpy.random import randn
import numpy as np
np.random.seed(12345)
np.set_printoptions(precision=4)
from pandas import Series, DataFrame
import pandas as pd
```

In [ ]:
```
%pwd
```

Series

- Series (http://pandas.pydata.org/pandas-docs/stable/dsintro.html#series) is a one-dimensional labeled array holding any data type
- Represent a row or a column in a table
- A series can be created from a list, a dict, a NumPy array, etc.

```
s = Series(list)
s = Series(dict)
s = Series(1D-array)
```

In [ ]:
```
s1 = Series([4, 7, -5, 3])
s1
```

In [ ]:
```
# Create a series from a dict
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
s3 = Series(sdata)
s3
```

Index

- The axis labels are collectively referred to as the index.
  - Elements are accessed by their indexes
    - both single element or range of elements
  - Individual index cannot be changed, but the set of index can be replaced

```
s.index = newIndexType
```

  - Default index is int from 0, 1, 2, ...

In [ ]:
```
s1.values
```

In [ ]:
```
print(s1.index)
```

In [ ]:
```
s1[2]
```

In [ ]:
```
#s2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
s2=s1
s2.index=['d', 'b', 'a', 'c'] # change index object
s2
```

In [ ]:
```
s2.index
```

In [ ]:
```
s2['a']
```

In [ ]:
```
s2['d'] = 6
s2[['c', 'a', 'd']]
```

Functions on Seriase

- Numpy functions can be applied to Series
  - Boolean expression:

    ```
    s>2, s!=0
    ```

  - Alrithmetic operations:

    ```
    s1+s2, s**3
    ```

  - Boolean functions

    ```
    s.isnull(), s.notnull()
    ```

In [ ]:
```
s2[s2 > 0] # get positive values
```

In [ ]:
```
s2 * 2  # double all values
```

In [ ]:
```
np.exp(s2) # exponential e to s2[x]
```

In [ ]:
```
'b' in s2 # check for an index
```

In [ ]:
```
'e' in s2
```

In [ ]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
s4 = Series(sdata, index=states) # change index object
s4
```

In [ ]:
```
# test for null values
pd.isnull(s4)
```

In [ ]:
```
pd.notnull(s4)
```

In [ ]:
```
s4.isnull()
```

In [ ]:
```
s3
```

In [ ]:
```
s4
```

In [ ]:
```
s3 + s4
```

Naming the Index and the Values

In [ ]:
```
# naming the index and the value
s4.name = 'population'
s4.index.name = 'state'
print(s4)
```

Exercise

- Create a Series called items from the following set of tuples

```
{('book', 53), ('chair', 4), ('table', None), ('printer', 1)}
```

- Create another Series called costs, which for each item list the total cost of that item, assuming each book costs $\$3$, each chair $\$10$, and printer costs $50.

DataFrame

- 2D table, with labeled rows and columns

```
df = DataFrame(data, columns=columnLabels,
                     index=indexLabels)
```

- data object can be Dict of 1D ndarrays, lists, dicts, or Series, 2-D numpy.ndarray, and many other data objects
- columnLabels and indexLabels are optional list-like objects

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object.

Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass index (row labels) and columns (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

Create DataFrame

- Basic way is to provide a dict to the DataFrame constructor
  - Keys are names of columns
  - Values are lists of values for the columns. The lists should have the same size
- Can copy from existing DataFrame
- Can also change the column's names, ordering, etc.

In [ ]:
```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
frame
```

Access Elements in a DataFrame

```
by element:   df.loc[row, col]
by column(s): df[col] or df.col, or
              df[[col1, col2, ..]]
by row(s):    df.loc[row] or df.loc[[row1, row2,...]]
```

- Single row or column is returned as a Series
- Access can be combined with assignment to change values

In [ ]:
```
frame.loc[1, 'year']
```

In [ ]:
```
frame[['pop', 'year']]
```

In [ ]:
```
frame.year
```

In [ ]:
```
frame.loc[1]
```

In [ ]:
```
frame.loc[[2, 3]]
```

Alter the Structure of a DataFrame

- Change index type
- Change ordering of the columns and rows
- Add new columns and rows with or without data values
- Swap rows and columns, that is, transpose a table
- Delete/Drop columns and rows

In [ ]:
```
# reorder columns
frame=frame[['year', 'state', 'pop']]
frame
```

In [ ]:
```
# Add a new row
frame.loc[5] = Series({'year':2018, 'state':'Texas', 'pop':1.9})
frame
```

In [ ]:
```
# Add a new column
frame['new_col'] = Series(['a', 'b', 'c', 'd'])
frame
```

In [ ]:
```
# Table transpose
frame = frame.T
frame
```

In [ ]:
```
# Add two new columns 'debt' and 'columns'
# without adding data values, results in NaN values
# also change index
# created a new frame
frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt', 'columns'],
                   index=['one', 'two', 'three', 'four', 'five'])
frame2
```

Pitfall: Using 'columns' as a name of a column

In [ ]:
```
frame2.columns
```

In [ ]:
```
# Column named "columns" is different from the attribute named columns
# So, do not name column using "columns", too confussing!
frame2['columns']
```

Assign Data Values into Columns

Can assign

- the same value for all rows
- consecutive values for next row
- specific values for selected rows
- value based on values in other columns of the same row

In [ ]:
```
# Assign the same value to every row in the one column
frame2['debt'] = 16.5
frame2
```

In [ ]:
```
# Assign consecutive values to consecutive rows
frame2['debt'] = np.arange(5.)
frame2
```

In [ ]:
```
# Assign a set of values to a set of selected rows
val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
frame2['debt'] = val
frame2
```

In [ ]:
```
# Add a new column with logic values based on a test on another column
frame2['eastern'] = frame2.state == 'Ohio'
frame2
```

Remove Selected Column(s)

```
del df[columns]
```

In [ ]:
```
del frame2['eastern']
del frame2['columns']
print(frame2.columns)
frame2
```

Other Ways to Create DataFrames

In [ ]:
```
# Use a nested dict
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

In [ ]:
```
frame3 = DataFrame(pop)
frame3
```

In [ ]:
```
# Transpose a table
frame3.T
```

In [ ]:
```
# Selected a new set of rows, resulted in NaN values
DataFrame(pop, index=[2001, 2002, 2003])
```

In [ ]:
```
# Use a slice of data from a DataFrame to create a new DataFrame
pdata = {'Ohio': frame3['Ohio'][:-1],
         'Nevada': frame3['Nevada'][:2]}
DataFrame(pdata)
```

In [ ]:
```
# Assign names to index and columns
frame3.index.name = 'year'; frame3.columns.name = 'state'
frame3
```

Get data values from a DataFrame into a NumPy array

In [ ]:
```
frame3.values
```

In [ ]:
```
frame2.values
```

Exercise

Consider the following DataFrame

```
df = pd.DataFrame(np.array(randn(28)).reshape(7, 4),
                  columns=['a', 'b', 'c', 'd'])
```

- Add a new column 'e' that contains the product of the values in columns 'b' and 'd'
- Append the following tupe as new row in the table

```
(1, 2, 3, 4, np.NaN)
```

Index Objects

- By default, index is a range starting at 0, but can also be other type objects.
- Index values cannot be changed.
- Can check index type or whether a specific value is in the index

In [ ]:
```
obj = Series(range(3), index=['a', 'b', 'c'])
index = obj.index
index
```

In [ ]:
```
index[1:]
```

In [ ]:
```
# Index does not support mutable operations
#index[1] = 'd'
```

In [ ]:
```
index = pd.Index(np.arange(3))
obj2 = Series([1.5, -2.5, 0], index=index)
obj2.index is index
```

In [ ]:
```
frame3
```

In [ ]:
```
'Ohio' in frame3.columns
```

In [ ]:
```
2003 in frame3.index
```

Essential Functionality

- Provided for both Series and DataFrame
    - Reindexing and renameing
    - Data selection
    - Arithmetic, function mapping
    - Sorting, ranking
    - Descriptive statistics
    - Hierarchical indexing

Reindexing

```
df.reindex(index=newIndexObj,
           method=fillMethod,
           columns=newColumnsObj,
           fill_value=defaultValue)
```

- If new index/columns has more rows/columns, NaN is filled by default
- fill_value of methods (ffill or bfill) can supply other fill values

Reindexing vs Renaming Index

- If reindex introduces new labels, new columns or rows will be created, with NA values
- Renaming will simply change the labels without adding new columns or rows

```
df.coulumns=[newLabels]
df.rename(columns = {old : new,...},
          index = {old : new, ...},
          inplace = True)
```

- Index is for rows and columns is for columns
- Once created, index labels can not change, but the index object can be changed
  - use index=
  - or use reindex() function, which can change both the index and the columns
- If the new index have more rows than existing rows, one can add new values to new rows
  - if nothing is done, NaN (also called a null value) will be added
  - can specify fill_value
  - can specify fill method, such as ffill or bfill, for forward or backward fill values

In [ ]:
```
obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj
```

In [ ]:
```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj2
```

In [ ]:
```
# Can provide default data values for new rows
obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
```

In [ ]:
```
obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj3
```

In [ ]:
```
obj3.reindex(range(6), method='ffill')
```

In [ ]:
```
frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
                  columns=['Ohio', 'Texas', 'California'])
frame
```

In [ ]:
```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2
```

In [ ]:
```
states = ['California','Texas', 'Utah']
frame.reindex(columns=states)
```

In [ ]:
```
#frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
#               columns=states)
```

In [ ]:
```
frame.loc[['a', 'b', 'c', 'd'], states]
```

Exercise

Given the following DataFrame, change the labels so that index becomes 'a', 'b', and 'c' and the columns arer labeled by 'California','Texas', and 'Utah'

```
frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
          columns=['Ohio', 'Texas', 'California'])
```

Dropping Entries from an Axis

- Rows and/or columns can be dropped (removed from the table)

```
df.drop(labels)          drop rows
dfdrop(labels, axis=1)   drop columns
```

- Notice drop() and many other functions do not change the original table, only a copy is producecd

In [ ]:
```
obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
new_obj = obj.drop('c') # remove a row
print(obj)
print(new_obj)
```

In [ ]:
```
obj.drop(['d', 'c'])
```

In [ ]:
```
data = DataFrame(np.arange(16).reshape((4, 4)),
                 index=['Ohio', 'Colorado', 'Utah', 'New York'],
                 columns=['one', 'two', 'three', 'four'])
data
```

In [ ]:
```
#remove rows
data.drop(['Colorado', 'Ohio'])
```

In [ ]:
```
# remove column
data.drop('two', axis=1)
```

In [ ]:
```
data.drop(['two', 'four'], axis=1)
```

Data Selection

- Data can be selected using indexing or Boolean expression
- For Series,

```
s[labels] refers rows
```

- For DataFrame,

```
df[labels] refers columns,
df.loc[[rows],[columns]] or df.ix[[rows],[columns]]
or df.iloc[[rows],[columns]]) refers a cell
```

In [ ]:
```
obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
print(obj)
print(obj['b'])
```

In [ ]:
```
obj[1]
```

In [ ]:
```
obj[2:4]
```

In [ ]:
```
obj[['b', 'a', 'd']]
```

In [ ]:
```
obj[[1, 3]]
```

In [ ]:
```
obj[obj < 2]
```

In [ ]:
```
obj['b':'c']
```

In [ ]:
```
obj['b':'c'] = 5
obj
```

In [ ]:
```
data = DataFrame(np.arange(16).reshape((4, 4)),
                 index=['Ohio', 'Colorado', 'Utah', 'New York'],
                 columns=['one', 'two', 'three', 'four'])
data
```

In [ ]:
```
data['two']
```

In [ ]:
```
data[['three', 'one']]
```

In [ ]:
```
data[:2]
```

In [ ]:
```
data['three'] > 5
```

In [ ]:
```
data[data['three'] > 5].loc['Colorado', :]
```

In [ ]:
```
data.two = data.two.drop(['Colorado', 'Utah'])
data
```

In [ ]:
```
data < 5
```

In [ ]:
```
data[data < 5] = 0
```

In [ ]:
```
data
```

In [ ]:
```
data.loc['Colorado', ['two', 'three']]
```

In [ ]:
```
data.ix[['Colorado', 'Utah'], [3, 0, 1]]
```

In [ ]:
```
data.iloc[2]
```

In [ ]:
```
data.loc[:'Utah', 'two']
```

In [ ]:
```
data.ix[data.three > 5, :3]
```

Exercise

```
df = pd.DataFrame(np.arange(15).reshape(5, 3),
                  index=['a', 'b', 'c', 'd', 'e'])
```

- Reindex df, so that the columns are labeled by 'one', 'two', and 'three'
- Create another DataFrame df2, which contains the inner elements of df, that is, elements in the second to forth row and the second column.

Arithmatic Operations and Data Alignment

- Arithmatic operations between tables will be performed on elements on the same row and same column
- If could not align, NaN value will be returned, unless fill_value is specified

In [ ]:
```
s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
```

In [ ]:
```
s1
```

In [ ]:
```
s2
```

In [ ]:
```
s1 + s2
```

In [ ]:
```
df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                index=['Ohio', 'Texas', 'Colorado'])
df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df1
```

In [ ]:
```
df2
```

In [ ]:
```
df1 + df2
```

Arithmetic Methods with Fill Values

If fill values are to be used in place of NaN, arithmetic functions instead of operators such as, +, *, etc. must be used.

In [ ]:
```
df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))
df1
```

In [ ]:
```
df2
```

In [ ]:
```
df1 + df2
```

In [ ]:
```
df1.add(df2, fill_value=0)
```

In [ ]:
```
df1.reindex(columns=df2.columns, fill_value=0)
```

Operations Between DataFrame and Series

- A Series can be treated as a row or a column, depending on the index
- If used to represent a rwo, the index of the series must be aligned by name with the DataFrame's columns, otherwise, NaN value will be generated

In [ ]:
```
arr = np.arange(12.).reshape((3, 4))
arr
```

In [ ]:
```
arr[0]
```

In [ ]:
```
arr - arr[0]
```

In [ ]:
```
frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])
series = frame.iloc[0]
frame
```

In [ ]:
```
series
```

In [ ]:
```
frame - series
```

In [ ]:
```
series2 = Series(range(3), index=['b', 'e', 'f'])
frame + series2
```

In [ ]:
```
series3 = frame['d']
frame
```

In [ ]:
```
series3
```

In [ ]:
```
frame.sub(series3, axis=0)
```

Function Application and Mapping

- Numpy functions and user defiend functions can also applied to data using

```
      df.apply(func): apply func to columns or rows
   df.applymap(func): apply func to elements in DataFrame
        df.map(func): apply func to elements in Series
```

In [ ]:
```
frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

In [ ]:
```
frame
```

In [ ]:
```
np.abs(frame)
```

In [ ]:
```
f = lambda x: x.max() - x.min()
```

In [ ]:
```
frame
```

In [ ]:
```
frame.apply(f)
```

In [ ]:
```
frame.apply(f, axis=1)
```

In [ ]:
```
def f(x):
    return Series([x.min(), x.max()], index=['min', 'max'])
frame.apply(f)
```

In [ ]:
```
# Define a lambda function to format data to have 2 precision digits
format = lambda x: '%.2f' % x
frame.applymap(format)
```

In [ ]:
```
frame['e'].map(format)
```

Exercise

What is the output?

```
df = pd.DataFrame(np.randn(3,4), columns=['a', 'b', 'c', 'd'])
s = pd.Series([2, 3, 0, 5], index=['a', 'e', 'c', 'd'])
df * s
```

Sorting and Ranking

- Sort by row index, column names, or values in columns

```
df.sort_index(),  df.sort_index(axis=1)
df.sort_value(by=columns)
```

- The rank of a value is its position (from 1 to n) in a sorted list

```
df.rank()       within each column
df.rank(axis=1) within each row
```

In [ ]:
```
obj = Series(range(4), index=['d', 'a', 'b', 'c'])
print(obj)
obj.sort_index()
```

In [ ]:
```
frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
                  columns=['d', 'a', 'b', 'c'])
frame.sort_index()
```

In [ ]:
```
frame.sort_index(axis=1)
```

In [ ]:
```
frame.sort_index(axis=1, ascending=False)
```

In [ ]:
```
frame
```

In [ ]:
```
obj = Series([4, 7, -3, 2])
obj.sort_values()
```

In [ ]:
```
obj = Series([4, np.nan, 7, np.nan, -3, 2])
obj.sort_values()
```

In [ ]:
```
frame = DataFrame({'b': [4, 7, -3, 2], 'a': [1, 0, 0, 1]})
frame
```

In [ ]:
```
frame.sort_values(by='a')
```

In [ ]:
```
frame.sort_values(by=['b', 'a'])
```

Ranking with Duplicate Values

Equal values are assigned a rank based on method:

```
df.rank(method=methodName)
method names:
  average: average rank of group (default)
      min: lowest rank in group
      max: highest rank in group
    first: ranks assigned in order they appear
           in the array
    dense: like 'min', but rank always increases
           by 1 between groups
```

In [ ]:
```
obj = Series([7, -5, 7, 4, 2, 0, 4])
print(obj.sort_values())
obj.rank()
```

In [ ]:
```
obj.rank(method='first')
```

In [ ]:
```
obj.rank(ascending=False, method='max')
```

In [ ]:
```
frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
                   'c': [-2, 5, 8, -2.5]})
frame
```

In [ ]:
```
frame.rank(ascending=False, method='max')
```

In [ ]:
```
frame.rank(axis=1)
```

Axis Indexes with Duplicate Values

- The row index may contain dulicate values

In [ ]:
```
obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
obj
```

In [ ]:
```
obj.index.is_unique
```

In [ ]:
```
obj['a']
```

In [ ]:
```
obj['c']
```

In [ ]:
```
df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
df
```

In [ ]:
```
df.loc['b']
```

Exercise

Write a code to sort the following DataFrame so that only positive elements are remaining and rows are sorted in descending order by column 'b'.

```
df = DataFrame(np.random.randn(4, 3), columns=['a', 'c', 'b']))
```

Summary and Descriptive Statistics

- DataFrame funcitons:

```
count(), mean(), sum(), describe(), etc.
```

  - Use axis=1 to change to statistics for rows
  - Can specify how to handle NaN values

In [ ]:
```
df = DataFrame([[1.4, np.nan], [7.1, -4.5],
                [np.nan, np.nan], [0.75, -1.3]],
               index=['a', 'b', 'c', 'd'],
               columns=['one', 'two'])
df
```

In [ ]:
```
df.sum()
```

In [ ]:
```
df.sum(axis=1)
```

In [ ]:
```
df.mean(axis=1, skipna=False)
```

In [ ]:
```
df.idxmax()
```

In [ ]:
```
df.cumsum()
```

In [ ]:
```
df.describe()
```

In [ ]:
```
obj = Series(['a', 'a', 'b', 'c'] * 4)
obj.describe()
```

Unique Values, Value Counts, and Membership

```
pd.Series.unique()    : list unique values
pd.value_counts()     : unique values and their counts
pd.Series.isin(aSet)  : whether each value is in a set
```

- These functions only work for a Series (a row or a column)

In [ ]:
```
#obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
obj = DataFrame(np.array(['c', 'a', 'd', 'a', 'a', 'b', 'b',
                          'c', 'c']).reshape((3, 3)),
               columns=['c1', 'c2', 'c3'],
               index=['r1', 'r2', 'r3'])
```

In [ ]:
```
# find unique values
uniques = obj['c2'].unique()
uniques
```

In [ ]:
```
obj['c2'].value_counts()
```

In [ ]:
```
pd.value_counts(obj['c2'].values, sort=False)
```

In [ ]:
```
# Test values for membership in a set
mask = obj['c2'].isin(['b', 'c'])
mask
```

In [ ]:
```
# Retrieve values that are members of a set
obj['c2'][mask]
```

In [ ]:
```
data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
                  'Qu2': [2, 3, 1, 2, 3],
                  'Qu3': [1, 5, 2, 4, 4]})
data
```

In [ ]:
```
# for each value in table count for each column
result = data.apply(pd.value_counts).fillna(0)
result
```

Exercise

Write a line of code to sum the data elements for each column of a DataFrame that are between -0.5 and +0.5 inclusive, without counting null values.

Test it on the following DataFrame.

```
df = DataFrame(np.random.randn(5, 3), columns=['a', 'c', 'b'])
```

Correlation and Covariance

- Use functions on columns

```
      cov(): find covariance
     corr(): find correlation
 corrwith(): correlation between columns of
             different DataFrames
```

- More will come late when we review Statistics

In [ ]:
```
frame = pd.DataFrame(np.random.randn(1000, 5),
                     columns=['a', 'b', 'c', 'd', 'e'])
frame
```

In [ ]:
```
# for each column, compute (value(t)-value(t-k))/value(t-k)
frame.pct_change(periods=3)
```

In [ ]:
```
# covariance between pairs of columns
frame.cov()
```

In [ ]:
```
# correlation between columns 'b' and 'd'
frame.b.corr(frame.d)
```

In [ ]:
```
frame.corr()
```

In [ ]:
```
# correlation between each column and column 'c'
frame.corrwith(frame.c)
```

In [ ]:
```
index = ['a', 'b', 'c', 'd', 'e']
columns = ['one', 'two', 'three', 'four']
df1 = pd.DataFrame(np.random.randn(5, 4), index=index, columns=columns)
df2 = pd.DataFrame(np.random.randn(4, 4), index=index[:4], columns=columns)
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:
```
df1.corrwith(df2)
```

In [ ]:
```
df2.corrwith(df1, axis=1)
```

Handling Missing Values

- Missing values can be represented by
  - NaN or np.nan for numbers
  - None for other type of object
- Functions

```
isnull(): test for null (or missing) values
dropna(): remove null values
fillna(): replace null value by a fill_value
```

- Different method can be used to determine the fill_value

In [ ]:
```
string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])
string_data
```

In [ ]:
```
string_data.isnull()
```

In [ ]:
```
string_data[0] = None
print(string_data)
string_data.isnull()
```

Drop Missing Values

```
df.dropna(how=methodName)
methods: 'all'
```

In [ ]:
```
from numpy import nan as NA
data = Series([1, NA, 3.5, NA, 7])
data.dropna()
```

In [ ]:
```
data[data.notnull()]
```

In [ ]:
```
data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
                  [NA, NA, NA], [NA, 6.5, 3.]])
data
```

In [ ]:
```
# remove any row with a null value
data.dropna()
```

In [ ]:
```
# remove rows that contain only null values
data.dropna(how='all')
```

In [ ]:
```
# add a new column of null values
data[4] = NA
data
```

In [ ]:
```
# remove columns with only null values
data.dropna(axis=1, how='all')
```

In [ ]:
```
df = DataFrame(np.random.randn(7, 3))
df.iloc[:4, 1] = NA; df.iloc[:2, 2] = NA
df
```

In [ ]:
```
# get rows with at least 3 non-null values
df.dropna(thresh=3)
```

Filling in Missing Values

```
df.fillna(values,
          method=methodName,
          limit=n)
```

In [ ]:
```
df.fillna(0)
```

In [ ]:
```
df.fillna({1: 0.5, 3: -1})
```

In [ ]:
```
# always returns a reference to the filled object
_ = df.fillna(0, inplace=True)
df
```

In [ ]:
```
df = DataFrame(np.random.randn(6, 3))
df.iloc[2:, 1] = NA; df.iloc[4:, 2] = NA
df
```

In [ ]:
```
df.fillna(method='ffill')
```

In [ ]:
```
df.fillna(method='ffill', limit=2)
```

In [ ]:
```
data = Series([1., NA, 3.5, NA, 7])
data.fillna(data.mean())
```

Exercise

Write a function which takes a DataFrame as a parameter and replace each missing value (NaN or None) with the mean of its column.

Hierarchical Indexing

- Both the row index and column names can be formed from multiple levels.

```
  index=[[level1], [level2],...]
columns=[[level1], [level2],...]
```

- Each index label is a tuple, for example (label1, label2, ...)
- More flexible than a single level index
- The index levels can be changed, reordered, stack/unstack, etc.

```
df.unstack(), df.stack()
```

In [ ]:
```
data = Series(np.random.randn(10),
              index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
                     [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])
data
```

In [ ]:
```
data.index
```

In [ ]:
```
data['b']
```

In [ ]:
```
data['b':'c']
```

In [ ]:
```
data.loc[['b', 'd']]
```

In [ ]:
```
data[:, 2]
```

In [ ]:
```
data.unstack()
```

In [ ]:
```
data.unstack().stack()
```

In [ ]:
```
frame = DataFrame(np.arange(12).reshape((4, 3)),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=[['Ohio', 'Ohio', 'Colorado'],
                           ['Green', 'Red', 'Green']])
frame
```

In [ ]:
```
frame.index.names = ['key1', 'key2']
frame.columns.names = ['state', 'color']
frame
```

In [ ]:
```
frame['Ohio']
```

MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']], names=['state', 'color'])

Reordering and Sorting Index Levels

```
df.swaplevel(level1, level2)
df.sort_index(level=n)
```

In [ ]:
```
frame.swaplevel('key1', 'key2')
```

In [ ]:
```
frame.sort_index(level=1)
```

In [ ]:
```
frame.swaplevel(0, 1).sort_index(level=0)
```

Summary Statistics by Level

In [ ]:
```
frame.sum(level='key2')
```

In [ ]:
```
frame.sum(level='color', axis=1)
```

Use Data Columns As Hierarchical Index

```
df.set_index([col1, col2, ...], drop=True)
df.reset_index
```

In [ ]:
```
frame = DataFrame({'a': range(7), 'b': range(7, 0, -1),
                   'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
                   'd': [0, 1, 2, 0, 1, 2, 3]})
frame
```

In [ ]:
```
frame2 = frame.set_index(['c', 'd'])
frame2
```

In [ ]:
```
frame.set_index(['c', 'd'], drop=False)
```

In [ ]:
```
frame2.reset_index()
```

Other pandas Topics

Many additional functions and topics related to DataFrame will be cover in future lectures

Integer indexing

In [ ]:
```
ser = Series(np.arange(3.))
ser.iloc[-1]
```

In [ ]:
```
ser
```

In [ ]:
```
ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])
ser2[-1]
```

In [ ]:
```
ser.iloc[:1]
```

In [ ]:
```
ser3 = Series(range(3), index=[-5, 1, 3])
ser3.iloc[2]
```

In [ ]:
```
frame = DataFrame(np.arange(6).reshape((3, 2)), index=[2, 0, 1])
frame.iloc[0]
```